AFRL-RI-RS-TR-2007-265
Final Technical Report
December 2007

COOPERATIVE COMMUNICATION MECHANISM AND ARCHITECTURE FOR CROSS-LAYER COORDINATION

University of Texas

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. AC37/00

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2007-265 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/ /s/

DAVID KRZYSIAK
Work Unit Manager

WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DEC 2007	Final	Jun 06 – May 07

4. TITLE AND SUBTITLE

COOPERATIVE COMMUNICATION MECHANISM AND ARCHITECTURE FOR CROSS-LAYER COORDINATION

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8750-06-1-0091

5c. PROGRAM ELEMENT NUMBER
62304E

6. AUTHOR(S)

Scott Nettles and Christine Julien

5d. PROJECT NUMBER
AC37

5e. TASK NUMBER
UT

5f. WORK UNIT NUMBER
EX

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of Texas at Austin
101 E 27th St.
Austin TX 78712-1500

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/RIGC
525 Brooks Rd
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2007-265

12. DISTRIBUTION AVAILABILITY STATEMENT
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. WPAFB PA# 07-0650

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This work is focused on three areas all of which are related to the overall theme of cooperation in wireless networks: the relay channel, architectures for cross-layer cooperation, and abstractions that allow developers to express complex communications requirements in dynamic wireless networks. A protocol was designed based on "next hop neighborhoods" or loose routing which was tested in a network simulator. A new layer for the network stack was specified that focuses on the local neighborhood of nodes and beyond the one hop focus of the link layer.

15. SUBJECT TERMS
Communications Networks, Protocols, Network Stack, Cross-Layer Coordination

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			David Krzysiak
U	U	U	UL	42	19b. TELEPHONE NUMBER (*Include area code*) 315-330-7454

TABLE OF CONTENTS

LIST OF FIGURES

1.0 Introduction.

The following is the final report for contract FA8750-06-1-0091. The work focused on three areas all of which related to the overall theme of cooperation in wireless networks: the relay channel, architectures for cross-layer cooperation, and abstractions that allow developers to express complex communication requirements in dynamic wireless environments.

2.0 Area 1: The relay channel

The first area focuses on the relay channel. The relay channel is an information theoretic notion that captures in a mathematical way the fundamental ideas of one node acting as a relay for another. Limited practical work has been done on this idea. We proposed to develop a practical protocol based on these ideas using the idea of next hop neighborhoods. This work is being performed by Ketan Mandke as part of his work towards a dissertation in this area.

Our goal is to demonstrate practical results in this area using a working wireless node prototype. This will both complement the previous more theoretical work and also provide strong practical evidence of the applicability of these ideas. As such Ketan's efforts have focused on developing the Hydra prototype. Hydra is a prototype wireless node. The RF frontend is based on the GNU radio USRP, the Physical layer is implemented entirely in C++ in the GNU radio framework, and the MAC is also written in C++ using the Click framework. Ketan has been a major contributor to Hydra, both in writing code for the Physical layer and the MAC as well as debugging the implementation, and making major contributions to generating initial experimental results. He is one of the principle authors of the first Hydra overview [1] and also on an experience paper focusing on experimental issues [2]. He is currently implementing a relay-channel simulator.

The following is a discussion of Hydra and some of our early results. It is excerpted from [1], "Early Results on Hydra: A Flexible MAC/PHY Multihop Testbed," which was authored by Ketan Mandke, Soon-Hyeok Choi, Gibeom Kim, Robert Grant, Robert C. Daniels, Wonsoo Kim, Robert W. Heath, Jr., and Scott M. Nettles.

The impact of wireless communication and networking is obviously significant and growing. Today most communications take a single wireless hop from a node to a cell tower or access point. In future systems though, it is likely that architectures in which data may take multiple wireless hops will be widely adopted. For example, in next generation cellular systems two-hop relay architectures may be used to overcome coverage problems. Other prominent examples include mesh networks, ad hoc networks, many varieties of sensor networks, etc. In general, networks with multiple wireless hops couple networking issues more closely to those of wireless transmission, in part simply because of the implied need to make routing decisions over variable wireless channels. They also present opportunities for cross-layer optimizations. For example, the media

access control (MAC) layer may benefit from using special physical (PHY) layer techniques to avoid interference among nodes on a multihop path. Our research is broadly focused on such multihop networks where close coordination between the PHY and the network layers, especially the MAC, is important. Our ultimate goal is to harness the many increasingly sophisticated PHY techniques as well as a general understanding of wireless communication to create wireless networks that are better along a number of dimensions, including capacity, bandwidth, latency, reliability, and cost.

Most wireless networking research uses simulation as its major validation technique, but there is significant evidence that this is problematic. This is particularly true when sophisticated PHYs and MACs are used to communicate over real wireless channels. Our claim is that to achieve a high degree of realism, we must build working prototypes of the systems of interest and evaluate them using realistic channels, either by actually transmitting packets over the air, or at the least by using sophisticated channel emulation.

The Hydra testbed is a concrete realization of this approach. The most important goal of Hydra is to facilitate experimentation with state-of-the-art PHYs, MACs, and other network protocols and their interactions, and to do so on working hardware over realistic channels. As such, Hydra places a premium on the flexibility of implementation of the PHY, MAC, and other network software. Each Hydra node is composed of an open-source reconfigurable radio frequency (RF) front-end connected to a general-purpose computer in which the PHY, MAC, and other network functionality are implemented in software. In particular, we use the Universal Software Radio Peripheral (USRP) board developed by Ettus Research to implement the RF front-end of Hydra, our PHY is implemented in C++ using the GNU Radio framework, and our MAC is implemented in C++ using the Click modular router framework. Click also provides wireless routing and a connection to the Linux TCP/IP stack, which facilitates end-to-end testing.
The use of C++ and several open-source frameworks promotes a further goal of making Hydra accessible to researchers who may not wish or be able to implement high performance hardware. The low cost of our software-defined approach means that it is feasible to have a significant number of Hydra nodes; and we believe the low cost and open-source nature of the Hydra platform will facilitate duplication of Hydra outside of UT Austin. Our hope is that the end result will be a system that we and other researchers can use effectively to validate a wide variety of cross-layer algorithms and protocols.

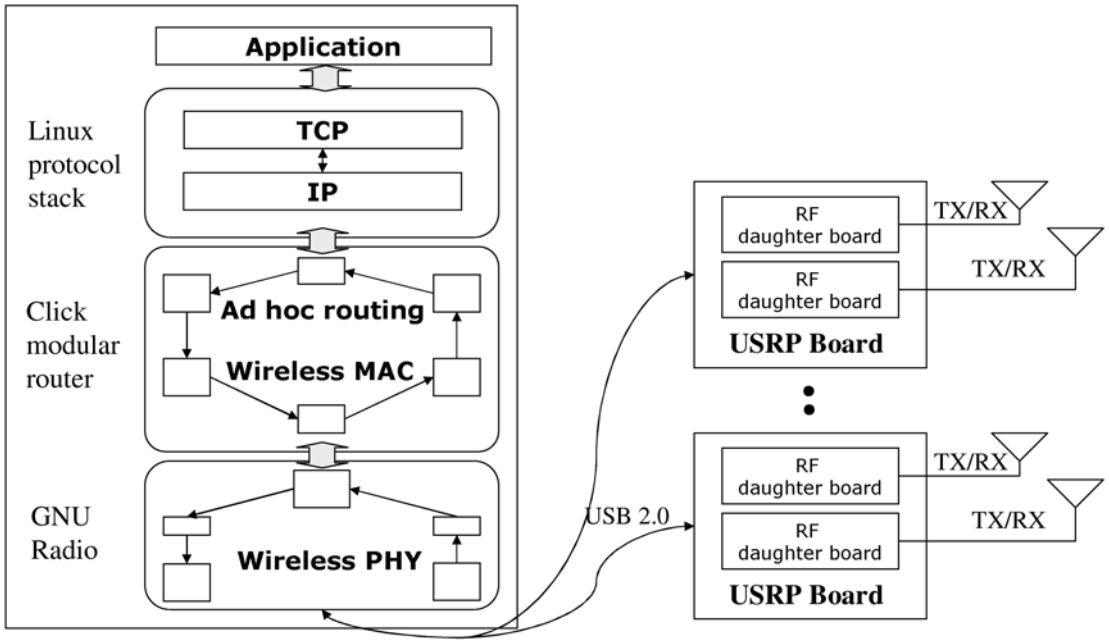

Figure 1: Block diagram of a Hydra node.

2.1 System Architecture and Implementation

This section describes the hardware and software used to implement Hydra. Figure 1 shows the block diagram of a Hydra node. Each node is composed of a general purpose PC (GPP) and a programmable RF front-end. The RF front-end performs some basic filtering as well as upconversion and downconversion and connects to the GPP using USB 2.0. All other aspects of Hydra are implemented in software on the GPP. Although an FPGA or ASIC implementation would provide better performance, using a high-level programming language (C++) on a GPP makes our system easy to change, flexible, and programmable by researchers with little or no `hardware' background. Additionally, the cost for each Hydra node is only $1250 USD plus the cost of a GPP.

2.2 THE RF FRONT-END

Hydra's RF front-end is the USRP board developed by Ettus Research for use with GNU Radio. The USRP is comprised of a baseband board and up to two RF daughter cards. The baseband board is composed of a USB 2.0 controller, a million gate FPGA, and multiple high-speed DA and AD converters. The FPGA provides control functionality, FIFO data queuing, as well as decimation/interpolation filtering. The baseband board's four high-speed DA and AD converters operate at 128 and 64 mega-samples-per-second respectively. The DA/AD converters also have programmable gain amplifiers controlled by the FPGA. All of the control and data manipulation functionality of the baseband board can be controlled in software running on the GPP; and all data and control signals are communicated to the front-end over USB.

GNU Radio defines a flexible API for the front-end that also supports reconfiguration of the RF daughter cards. The daughter cards are frequency agile radio transceivers with programmable gain control. They can be swapped to allow transceiver operation in various frequency bands (e.g. 400-500 MHz, ISM band, etc.). Additionally, the front-end features a fully synchronous design that allows multiple daughter cards to be synchronized providing support for research in multiple antenna systems (i.e. MIMO systems).

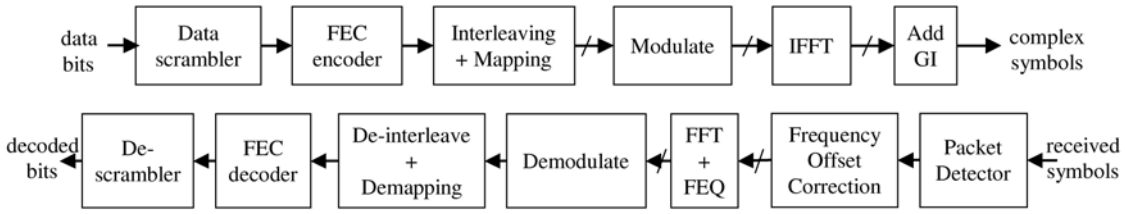

Figure 2: Block diagram of transmit and receive chain for a single-carrier OFDM physical layer based on IEEE 802.11a.

2.3 THE PHYSICAL LAYER

Hydra uses the GNU Radio framework for PHY implementation. This open-source software framework handles all hardware interfacing, multithreading, and portability issues, thus freeing the implementer to focus on the signal processing aspects of the software radio design. To compose a protocol in GNU Radio, users first build signal-processing blocks in C++ and then connect these blocks together using the Python language to form a flow graph. In addition to the relative ease of programming in C++, this approach makes it easy to transition blocks that are part of a PHY simulator into GNU Radio. In fact, the blocks that make up Hydra were initially developed and verified as part of a LabVIEW simulation. GNU Radio is also an attractive development platform because of its growing community of users, who range from amateur radio enthusiasts to university researchers.

To investigate advanced physical layer concepts (such as feedback in MIMO systems, OFDMA, and coding), Hydra's PHY supports OFDM and channel coding, and support for multiple antennas is being added. The PHY also has a very flexible interface to the MAC, implemented over UDP through a local IP connection. This low latency connection provides an efficient interface for cross-layer communication, and facilitates experiments in cross-layer design.

For our current experiments, a single-antenna OFDM physical layer (shown in Figure 2) was used. This PHY is based on the IEEE 802.11a physical layer. The prototype supports physical layer data rates of up to 5.4~Mbps. This rate is limited by the bandwidth of USB~2.0, but higher bandwidth interfaces are being investigated for future revisions of the front-end.

2.4 THE MAC LAYER

Hydra uses the Click framework for its MAC implementation. This software framework, developed at MIT, runs on a GPP and was originally created for building flexible and high performance routers. Similar to GNU Radio, packet-processing elements are coded in C++ and connected together using Click's own glue language. Elements can be flexibly configured to perform tasks for packet processing such as packet classification or scheduling, and then connected together in a flow graph to compose a protocol. Click not only handles memory management and scheduling for flow graph elements, but also allows users to select and modify various scheduling algorithms. As with GNU Radio, Click is also an attractive open-source development platform because of its growing community of users, which have used Click for a wide variety of applications including modular router design, ad hoc routing, and network coding.

We have implemented several random access MAC protocols for Hydra in Click, in particular CSMA/CA and the distributed coordination function (DCF) mode of IEEE 802.11. From this experience, it is clear that implementing a slotted MAC protocol for Hydra would be straightforward. As a proof of concept for Hydra's cross-layer design potential, we have also extended the current DCF MAC design to create a rate-adaptive MAC protocol based on RBAR. This cross-layer protocol will be used to investigate rate-adaptation in multihop networks.

2.5 OTHER PROTOCOL LAYERS

In Hydra, the wireless ad hoc networking protocols used to manage network connectivity are also implemented in Click. The code for these network protocols was contributed to the Click codebase by the creators of Grid, as well as other networking researchers. Since the MAC and networking protocols are both implemented in Click, they run together in their own address space, separate from GNU Radio and the TCP/IP protocol stack (running in the Linux user and kernel address space respectively). This parallelism may provide a performance improvement in multiprocessor environments.

Click features a simple tunneling mechanism that allows protocols to interface with the standard Linux TCP/IP stack. Thus, any application built on TCP/IP can be used with Hydra. This allows researchers using Hydra to easily test new wireless protocols with end-to-end application level experiments. Applications such as ping, ftp, and web sessions are used regularly to verify and debug the operation of Hydra.

2.6 A Cross-Layer Design Demonstration

To investigate how Hydra can facilitate exploration of cross-layer design, we implemented a rate-adaptive MAC protocol based on the Receiver Based Auto-Rate (RBAR) protocol and performed some preliminary experiments. We first discuss the protocol and then the details of the experiments.

2.7 THE RATE-ADAPTIVE MAC PROTOCOL

RBAR is a MAC protocol that uses the control messages of the DCF mode of IEEE 802.11 to perform opportunistic link level rate-adaptation. The goal is to use a higher rate when the wireless channel permits. The 802.11 DCF MAC consists of a four-way handshake. Request-To-Send (RTS) and Clear-To-Send (CTS) control messages reserve the ``floor'' around the sender and receiver prior to data transmission. After successfully decoding a data (DATA) packet, the receiver responds with an Acknowledgment (ACK) message. In RBAR, the PHY at the receiver uses the RTS message to estimate link quality between the sender and receiver. Then, the MAC extracts this link quality information from the PHY and determines the maximum rate that can be supported by the link. The receiver then piggybacks this rate information in the CTS response to the sender, which uses that rate to send the data. Cross-layer interactions are essential here, since the MAC has no direct way to determine link quality, while the PHY has no way to directly communicate with the sender.

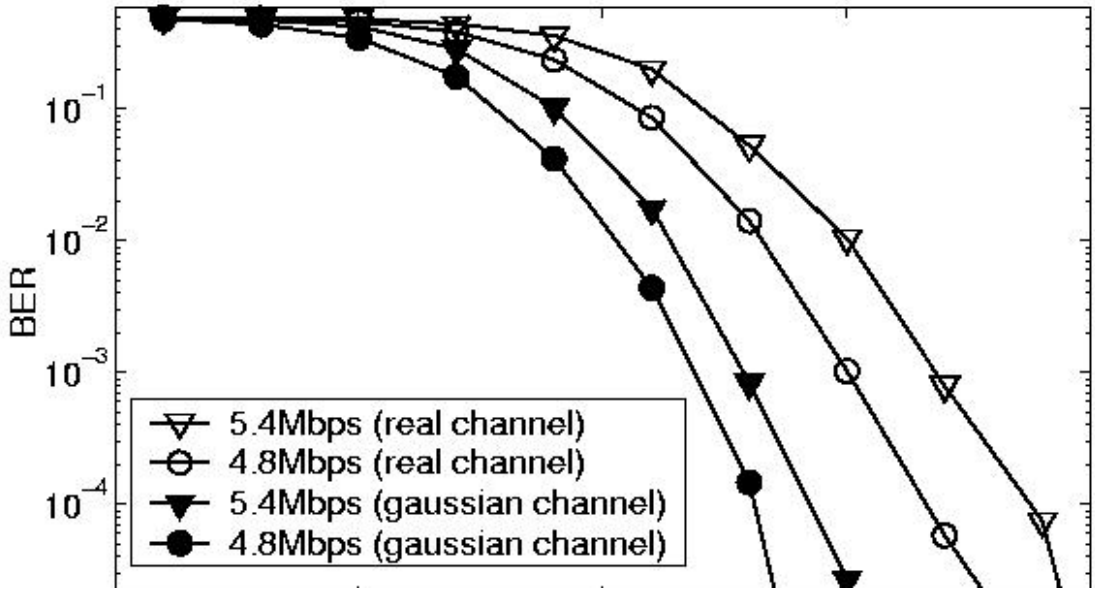

Figure 3: Calibration for the 4.8 Mbps and 5.4 Mbps data rates.

In Hydra, the PHY estimates the signal-to-noise ratio (SNR) for each packet and then piggybacks this information to the MAC layer on the decoded packet. The flexible interface between the MAC and PHY facilitates this cross-layer communication and can be extended to accommodate other types of MAC/PHY interaction needed, for example, in opportunistic sub-channel allocation in OFDMA or interference mitigation.

2.8 EXPERIMENTAL SETUP

In this simple experiment, our goal was simply to observe the rate adaptation process between two Hydra nodes. Hydra was operated in an indoor office environment. The two nodes were separated by a fixed distance. To change the wireless link quality, the transmit power of the sender node was varied over time. The MAC protocol implemented in Hydra responds to these changes in link quality by adapting the rate of the data transmission. The traffic pattern for this experiment consists of a stream of 1000 byte packets transmitted using UDP.

The MAC protocol selects the data rate for the transmission from the possible rates using a threshold policy. We ran a careful calibration procedure to select the proper SNR thresholds for this policy by measuring the bit-error rate (BER) performance of the system as it varied with SNR for each data rate. Figure 3 shows the results of this calibration procedure (a plot of SNR in dB versus average BER) for the data rates of 5.4~Mbps and 4.8~Mbps over both an emulated additive gaussian noise channel and an actual wireless channel. We set SNR thresholds for our MAC protocol in order to achieve an average BER below approximately 10^-5. For example, the thresholds for 4.8~Mbps and 5.4~Mbps were 28~db and 30~dB respectively.

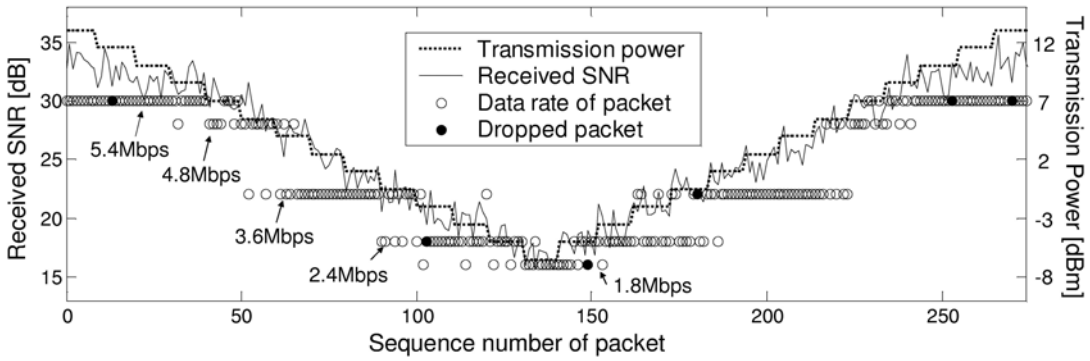

Figure 4: Trace for experiment with rate-adaptive MAC protocol.

Experimental Results

Figure 4 shows a trace of the results of this experiment. The X-axis is the packet sequence number and the Y-axis on the left is for the received SNR in dB and the Y-axis on the right is for the transmit power in dBm. Each packet is plotted at the SNR threshold for the rate at which it was transmitted. Open circles indicate that a packet was successfully received, while a closed circle indicates the packet was dropped.

In this trace, the transmit power was decreased and then increased in steps. We see that in general when changes in power cause the received SNR to cross a threshold, our MAC protocol adapts the data rate of the transmission as expected. We also see instances of packets that are transmitted at higher or lower data rates than most packets at the same power level. These are examples of how the algorithm can adapt to short-term fluctuations in the channel. Finally, upon investigating the cause of the dropped packets, we found they occurred when the channel worsened between the transmission of the RTS and the data, resulting in a packet sent at rate unsuitable for the channel. Thus even with this simple trace we see the complexities of studying such a protocol in the real world.

3.0 Area 2: Architectures for cross-layer cooperation

The second area of focus involved architectures for cross-layer cooperation. This work is primarily the work of Soon-Hyeok Choi as part of his dissertation proposal. This work is complete, with the exception of several additional case studies, which will be part of Soon-Hyeok's final dissertation, which is expected in the spring of 2008. We have created a taxonomy to capture the design space, a conceptual software architecture, and two concrete software architectures. As discussed above, there has been one case study, focusing on rate adaptation, implemented in the Hydra testbed to support this work. Two more case studies are planned. This work will appear in the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008 [3]. A preliminary version of this paper can be found on the web [4]. An additional paper, which uses the results of the first case study as the basis for a series of experiments involving rate-adaptation is in preparation [5].

The following is a discussion of our cross-layer architectures. It is excerpted from [4], "A Software Architecture for Cross-Layer Wireless Network Adaptations", which was authored by Soon-Hyeok Choi, Dewayne E. Perry and Scott M. Nettles.

The IP-based Internetwork has had an impact that its inventors could hardly have imagined. An important underlying key to the Internet's success is that its design and implementation is based firmly on a well-established architecture, the ``hourglass model''. The hourglass model defines a set of layers, each of which implements some aspect of the network, while leaving other aspects to higher levels. This architecture is a fundamental software engineering strategy to manage complexity in the design and implementation of a very large distributed hardware and software artifact.

Although, strictly speaking, the Internet is based on the hourglass model, for our purposes it is more useful to consider another layering model for networks, the OSI seven layer model, which for the four layers on which we will focus mirrors and somewhat refines the hourglass model. Figure 5 shows the layers and how they communicate in a conventional network implementation. The lowest layer is the physical layer (or PHY), which is responsible for actually sending data across a physical link. The PHY is the interface between the analog physical world and the digital world of data communications. Next, is the link/media access control layer (or MAC), which is responsible for managing communication for an individual link including coordinating when a sender is allowed to use a shared medium like the radio frequency (RF) spectrum. Next is the Network layer, which is responsible for connecting individual links into a multihop network that can deliver data between nodes that are not directly connected. For our purposes, the final layer is the Transport layer, which is responsible for coordinating end-to-end communication along the connections created by the Network layer. The key point is that each layer implements some key functionality with a well defined interface and leaves other functionality for the higher layers to implement. Taken together, we refer to the layers that make up the network as the stack. Figure 5 shows that in the conventional architecture each layer of the stack only communicates with the layer above and below it.

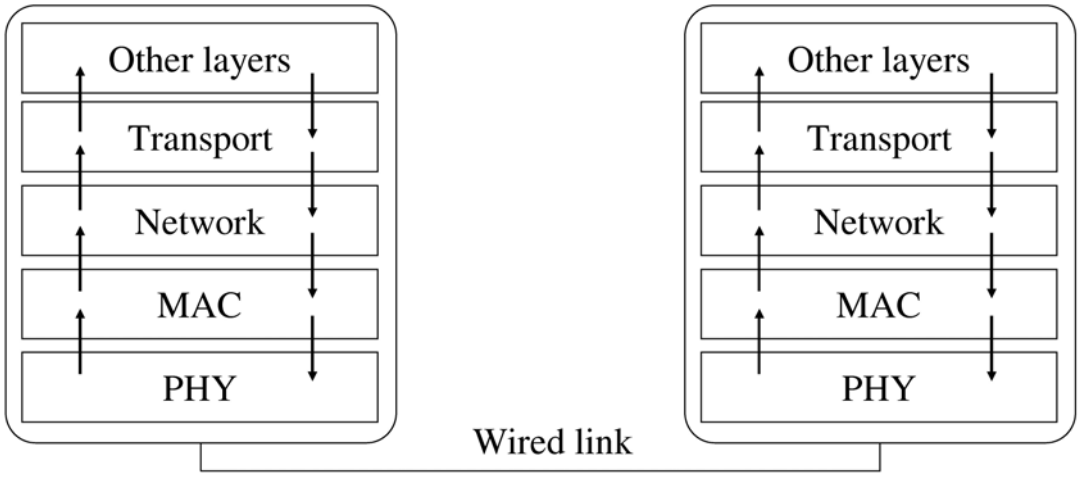

Figure 5: Conventional layered architecture

For wired networks, the layered architecture is remarkably successful and the key assumptions and abstraction boundaries work well. However, the introduction of wireless links based on RF communication has revealed that the abstractions are not as cleanly defined as one might expect (or hope) and that higher layers make unwarranted assumptions about lower ones. The classic example is when TCP is run over a wireless link. Because wireless links are subject to transmission errors, sometimes they drop a packet. Although TCP has no problem retransmitting the lost packet, it also interprets the drop as a sign that some node in the network is overloaded and dropped the packet to reduce its load. TCP reacts by slowing the rate at which it sends data. This is an incorrect choice when the drop is due to a transmission error and results from an invalid assumption that TCP makes about the reliability of the PHY, a layer that resides several levels down the stack. This is just one example where there needs to be enhanced communication across the layers and of late the area of ``cross-layer design'' has become a very active one.

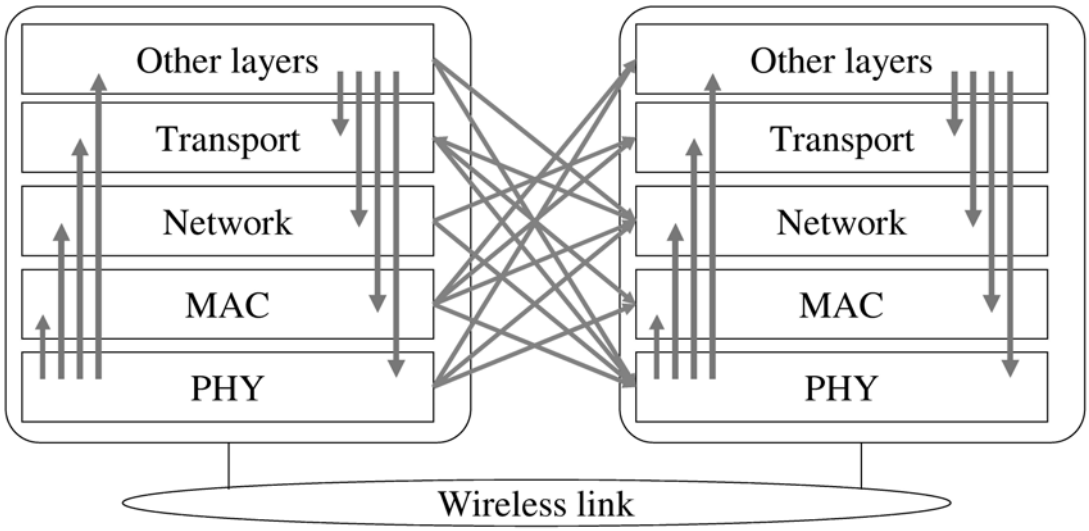

Figure 6: Cross-layer communication paths

Wireless ``links'' differ from wired ones in many ways. For example, they are lossy and their bandwidth varies with time. In fact, if the power level is changed, the set of nodes that are directly connected ``neighbors'' may even change. This leads to possible interactions between nonadjacent layers. For example, by changing power, a PHY property, the network layer might cause different routes to be discovered and used. Even adjacent layers may need to communicate in ways not possible in the current architecture. Later we will present an example where the MAC must obtain information from the PHY that is not part of normal packet processing. Figure 6 shows just some of the ways that information may need to cross between layers. Decision-making processes at any layer may need information from any other layer or even a set of other layers. As further shown in Figure 6, information might also be needed from other layers on other nodes, while the current architecture only allows communication between peer layers. Thus, in

general, cross-layer designs and implementations (or adaptations) may need almost arbitrary violations of the basic layering structure. Our goal is to develop an architecture that can accommodate this flexibility without destroying the current layered architecture with its advantages of modularity and robustness. Although our examples and research prototype focus on interaction between the MAC and PHY, our architecture accommodates other interactions, such as the Transport layer/MAC layer interactions needed to address the TCP over wireless problem.

Most of the work on cross-layer design has proceeded in an undisciplined way and has neglected the design and implementation advantages of the layered architecture. The result are systems that are basically spaghetti code with limited structure. Thus far there has been no general consideration of how to construct cross-layer adaptations in a systematic and modular manner. Our goal is to remedy this by providing a framework for building cross-layer protocols that maintains to a significant degree the advantages of modularity and abstraction found in the layered design. As such our focus in this paper is not on any particular cross-layer adaptation (except as an example), but rather on the software engineering issues that arise from the need to violate layering in general. Our strategy for achieving this is to first create a taxonomy that allows us to describe the design space of possible cross-layer adaptations. We then use this taxonomy as a framework to define a conceptual software architecture that allows us to implement adaptations within this space in a systematic way that preserves modularity. This architecture further motivates a concrete architecture that allows us to validate our concepts in a working wireless network prototype.

3.1 Motivating Examples

We developed and validated our taxonomy and architecture using a wide variety of example adaptations. We present two of these, cross-layer rate control and cross-layer protocol reconfiguration, for motivation here.

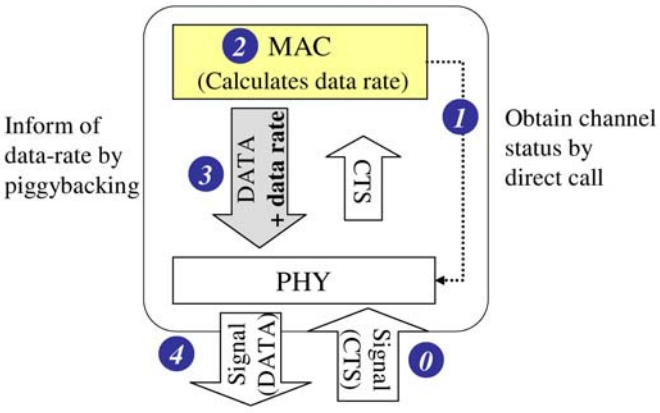

Figure 7: Rate control

For rate control the idea is simple. The maximum data rate depends on how good the RF connection (or channel) is between the sender and receiver. Ideally one sends at the highest rate possible, but the quality of the channel may change from packet to packet. One feasible solution arises because it is possible to measure the quality of the channel just before the data is sent. This is because in MACs such as the distributed coordination function (DCF) mode of IEEE 802.11, prior to data transmission there is an exchange of control messages between the sender and receiver to coordinate channel access. The sender first sends a request-to-send (RTS) to the receiver, which, if it is acceptable to send, replies with a clear-to-send (CTS). The sender's PHY receives the CTS and as a side effect can determine the quality of the channel. The sender's MAC can use this information to set the transmission rate of the immediately following data transmission. Cross layering arises because only the PHY can determine the channel state, but only the MAC knows which transmissions are control packets (and thus sent at a fixed low rate) and which are data (and thus candidates for sending at a higher rate).

Figure 7 shows the process in detail. In step 0, the PHY receives some data and decodes it estimating the channel quality as a side effect. In step 1, if the data was a CTS, the MAC makes a call into the PHY to get the channel information. In step 2, the MAC calculates the correct rate. In step 3, the MAC communicates the correct rate to the PHY by actually attaching the rate to the data packet, a process we refer to as piggybacking. Finally, in step 4, the PHY sends the packet at the specified rate.

A slight refinement of this example occurs if sender to receiver channel differs from the receiver to sender channel. In this case, we say that the channels are not reciprocal. Now we use the RTS to measure the channel quality. The MAC on the receiver then reads the channel quality from the PHY and piggybacks it on the CTS, which eventually results in the MAC on the sender obtaining the information. This is a simple case requiring internode communication.

The second main example is cross-layer protocol reconfiguration, which allows a node to switch from using one wireless protocol to another. This cross-layer adaptation has a different processing style from rate control. Suppose that we want to allow a mobile device to move from an IEEE 802.11 wireless network to a Bluetooth network. One approach is autonomous reconfiguration of the protocol layers by monitoring the wireless communication environment. Essentially the node listens for other nodes using different protocols and reconfigures itself to use the new protocols as needed. Such reconfigurations are not triggered by or coordinated with packet reception or transmission, but rather occur when a node detects that other nodes in its vicinity are using the different protocol.

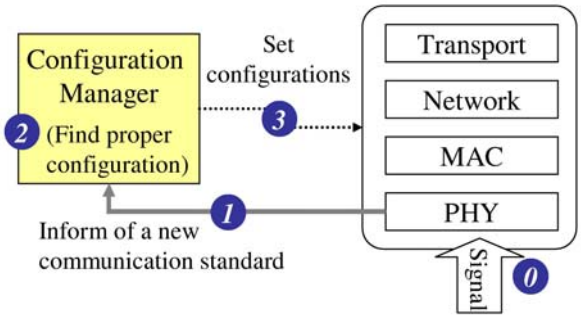

Figure 8: Protocol reconfiguration

Figure 8 shows the process in detail. The PHY is able to hear all the signals in the wireless environment and thus detects that the alternative communication standard is in use. In step 1, when the PHY detects a new standard, it informs the reconfiguration process. This notification serves as the trigger for reconfiguration. In step 2, the adaptation process finds the proper configurations of protocol layers that meet the new standard and, in step 3, it changes the configuration of the stack. Notice that the actual reconfiguration is coordinated outside of the protocol processing modules. Since the adaptation process requires global information about the protocol layers, using a global manager makes it easy to manage all the configuration parameters for all layers. Briefly, another example is the local agile routing protocol. Here, nodes monitor the channel conditions and exchange information between themselves to assist in creating advantageous routes. For our purpose, the key aspects of this protocol are: first, that it exchanges information between nodes without regard to whether the nodes are carrying data traffic or not, and, second, that it might exchange information with nodes that are more than one hop away.

3.2 Taxonomy

We use cross-layer rate control and protocol reconfiguration to motivate our taxonomy. Broadly speaking, the goal of developing this taxonomy is to characterize the complex and wide design space of cross-layer adaptations. As importantly, the taxonomy also defines a vocabulary that we can use to describe cross-layer adaptations and their implementation in our architecture. Finally, the taxonomy serves to guide the creation of our architecture itself.

Considering rate control, there are three primary constituents involved. First, at its heart there is the actual cross-layer adaptation process, here, the process that decides the rate given a channel condition. Second, some information is communicated across layers, here, the channel condition and the rate. Third, there are the delivery mechanisms that are used

to communicate the information to and from the process, here, a direct call to gather the channel state and piggybacking the rate on the data packet. Figure 9 shows our complete taxonomy, with these three basic categories, Information, Delivery Method, and Adaptation Process, making up the top-level.

Using rate control, we can partially refine each category. For information, there is one refinement based on the role of the information. The two subcategories are Status and Control, the roles of which are obvious. For Delivery Mechanism, this example illustrates a distinction based on the path of the information. Out-of-band information, such as the channel status, takes a path that is different from the actual packet, here, a procedure call from the MAC to the PHY. In-band data, such as the rate, takes the same path as the packet and can be piggybacked on the packet. Finally, we see two attributes of the Adaptation Process. The first is based on the time that the adaptation is performed. This example illustrates just one possibility in which the adaptation is Synchronous. This means that the adaptation is synchronized with the reception or transmission of a packet. Here, the rate calculation is triggered by receiving the CTS and must take place before transmitting the data. The second attribute is based on the location of the adaptation process. Again, this example only illustrates one possibility in which the adaptation is actually part of the MAC itself. We classify this adaptation as being Inside the protocol processor.

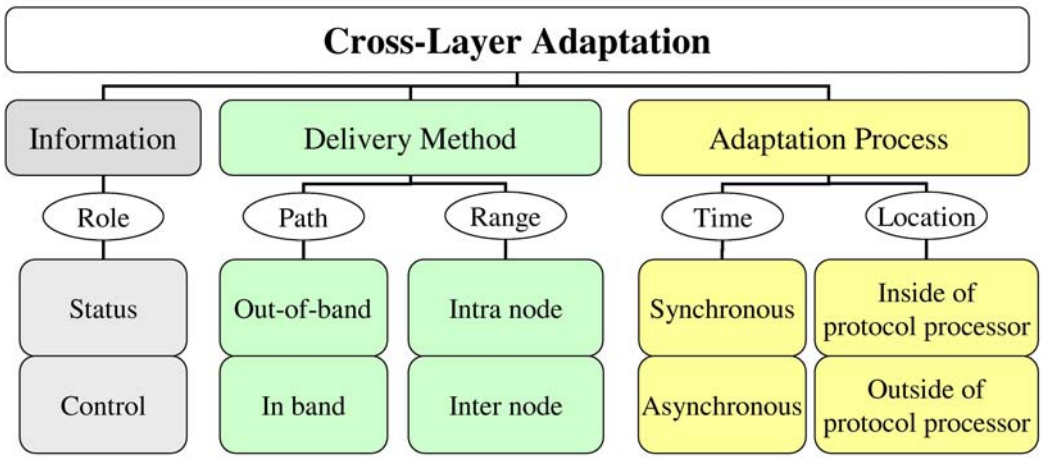

Figure 9: Taxonomy of cross-layer

Internode rate control gives rise to an additional distinction for the Delivery Method based on range. For basic rate control, the range is Intranode and for the internode version it is Internode. These distinctions are important because any mechanism that communicates between nodes over the RF link is inherently more expensive and failure prone than one that does not. For Intranode delivery, the distinction of In-band and Out-of-band is obvious as discussed above. But this classification is also applicable for Internode delivery. When the Internode version piggybacks the channel condition on the

CTS, this is In-band information that is piggybacked on an `existing' packet that is already being delivered to another node. In contrast, we might create a new packet to deliver Out-of-band information using an additional delivery path dedicated to cross-layer information.

Protocol reconfiguration allows us to complete our taxonomy by further refining the Adaptation Process. As a complement to Synchronous adaptations, the reconfiguration process is Asynchronous. This reconfiguration process is not coordinated with packet transmission or reception, but rather is triggered asynchronously when the PHY detects a new standard in its vicinity. Further, while rate control needs to finish its adaptation before the data transmission, the reconfiguration process can achieve its goal even if the actual time of adaptation is somewhat delayed after detection. Another significant refinement is that the reconfiguration process occurs Outside the protocol processor. Such adaptations are not part of the packet processing flow and thus might occur when the process needs to coordinate between a number of protocol layers, as might be needed in the control of quality of service or energy consumption.

3.3 An Architecture for Adaptations

Developing the taxonomy allowed us to describe the possible cross-layer adaptations succinctly. Our goal in developing an architecture is fundamentally to provide a set of mechanisms that can be used to implement a wide variety of cross-layer adaptations. Our main architecture is a conceptual one, which shows in general a set of components and their relationships in a system at a high-level abstraction. Thus our conceptual architecture helps us to understand how we can implement the desired adaptation processes using our framework and in practice serves as a reference model from which a variety of concrete architectures can be derived. This concrete architecture shows more detailed implementation issues that arise when we implement our framework on a specific wireless system. Thus, our architecture is a generic architecture, which can describe a range of cross-layer architectures.

We begin by presenting a series of high-level goals and requirements for the architecture. We then present some key architectural decisions. We then use the rate control and protocol reconfiguration examples to flesh out the details of the architecture. Finally we motivate a few aspects of the architecture that were not covered by the examples.

The most important goal of our architecture is to provide a set of mechanisms that support the implementation of all reasonable cross-layer adaptations described by our taxonomy. There are a number of secondary goals, which are fundamentally motivated by a desire to maintain the advantages of the existing layered architecture to the extent possible. The first goal is to preserve the modularity of existing protocol modules to the greatest extent possible. This is key, because otherwise we would be free to simply implement any cross-layer adaptation in an ad-hoc manner. The next goal is to allow

cross-layer adaptations to be implemented in as flexible and extensible a manner as possible as well as to facilitate implementing multiple adaptations in a single system. Finally, we want to allow our implementations to be portable to a variety of protocol implementations. For example, ideally if we implement rate adaptation for one particular system, it would be easy to move this implementation to some other system as long as it has the same underlying framework for cross-layer adaptations.

3.4 KEY ARCHITECTURAL DECISIONS

Figure 10 shows our taxonomy after we have applied two key architectural decisions. The first decision is simple. Although functionally different, the implementation of cross-layer information does not vary based on whether the data is used as status or control. Thus we can merge these two categories for the purpose of the architecture.

The other change, elimination of the ``Inside the protocol processor" location for the Adaptation Process requires more discussion. The motivation is simple, if we implement an adaptation as part of a protocol module, we will by necessity make changes that compromise system modularity. Furthermore because these changes will be intertwined with the implementation of the base protocol, flexibility, extensibility, and portability will also be compromised. Thus the key challenge in creating our architecture becomes a question of whether we can achieve our goal of comprehensive cross-layer adaptation support, without allowing substantive changes to the protocol modules themselves.

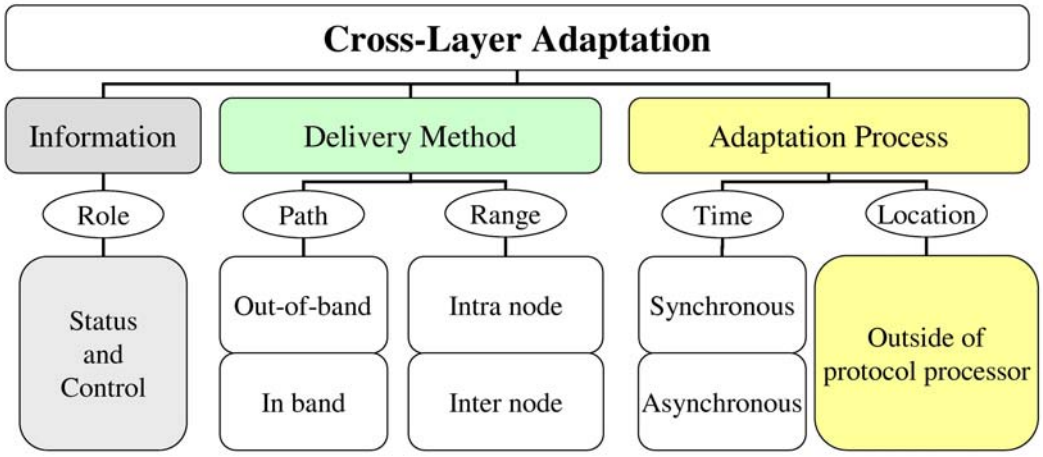

Figure 10: Refinement of our taxonomy

3.5 EXAMPLE DRIVEN DEVELOPMENT

Figures 11-13 shows the progression of high-level stages that are required to map our rate control example to our proposed architecture. We consider each stage in turn, explaining the architectural features required.

The first stage (Figure 11) shows the mechanisms needed to support a Synchronous adaptation process outside of the protocol module. Note that the rate control adaptation has been placed in a separate cross-layer module. A key requirement is that when the packet moves from the PHY to the MAC, the adaptation process must be notified if that packet is a CTS. Thus we see that in step 0, we have added a MAC-PHY interceptor module. This module is inserted between the two existing layers and provides each with the same interface and thus does not compromise our modularity goal. In general, this interceptor is a kind of connector, but it will be implemented as a ``shim" layer in the stack and so we do not group it with the other connectors discussed below. In step 1, the interceptor has detected a CTS and notifies the Synchronous event handler that connects the protocol module to the cross-layer module. Finally, in step 2 the event handler notifies the rate control process itself.

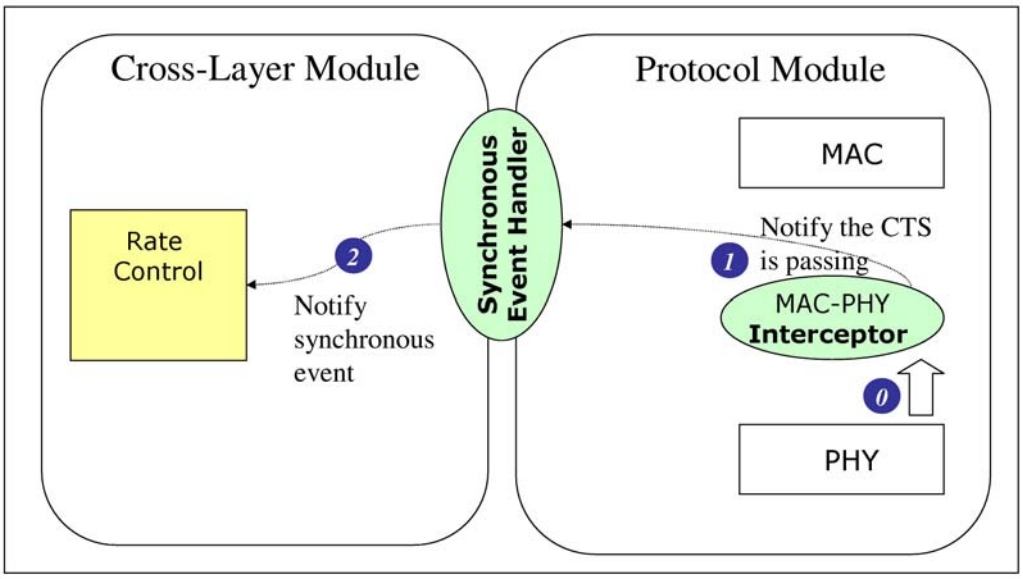

Figure 11: Components for Synchronous process

The second stage (Figure 12) shows the support needed for Out-of-band delivery. In step 1, the rate control process communicates to the Out-of-band connector that it needs the length of the packet and the channel status. Notice that unlike the case where the process is part of the MAC, it needs to access MAC as well as PHY information. In step 2, the connector communicates with the getLength and GetChannel adaptors attached to the MAC and PHY. The adaptation requires that we are able to query the protocol modules. By structuring these queries in terms of special adaptors, we are able to simplify changes to the protocol modules. A possible change in this case is to expose data length and channel information to allow the adaptors to access the information. Finally, in step 3, the rate control process calculates the new rate.

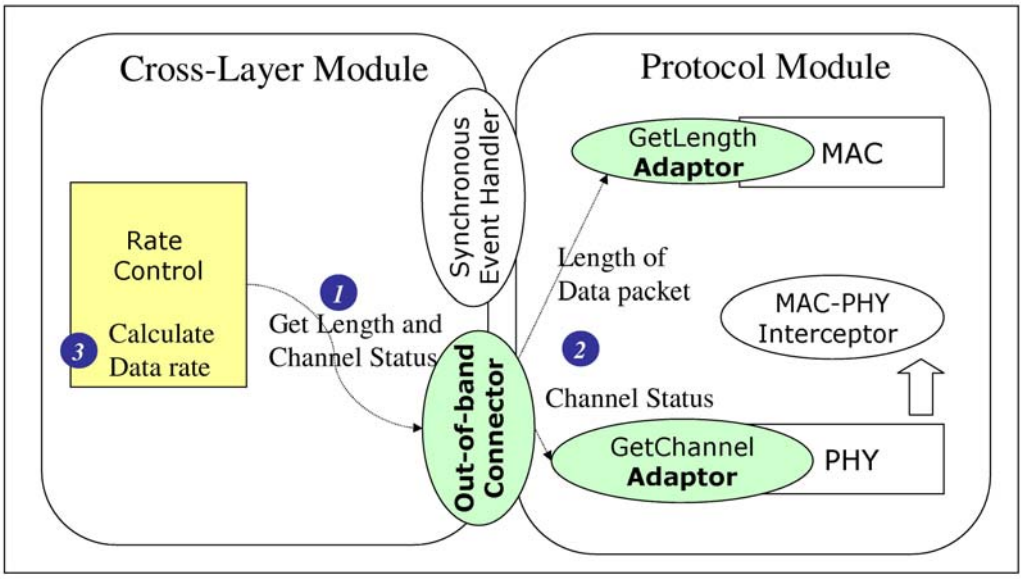

Figure 12: Components for Out-of-band delivery

The final stage (Figure 13) uses the existing mechanisms to complete the rate control process. In step 1 and 2, the interceptor notifies the rate control process that the data packet is being sent. In step 3 and 4, the rate control process sets the rate in the PHY using the setDataRate Adaptor.

To implement the protocol reconfiguration process, we can use most of the mechanisms introduced for rate control. In our example, a global configuration manager already performed the reconfiguration process outside of protocol module. Further, the existing Out-of-band connector allows the manager to read and update information inside protocol modules such as the communication standard stored in the PHY and the configurations of the protocol layers.

The only new requirement is triggering the reconfiguration process when the PHY detects a new communication standard, which is an Asynchronous adaptation process. One solution is to allow the PHY to notify the manager of the detected standard information. The problem with this active notification is that the PHY implementation needs to be changed to be aware of the manager, making the PHY dependent on the manager. To address this problem, we introduce an Asynchronous event handler that periodically polls the standards information stored in the PHY and triggers the manager when it detects a change. Such a polling mechanism only requires an additional Adaptor attached to the PHY and thus maintains the modularity of the PHY. Although the periodic polling may cause some bounded delay before detecting changes, the Asynchronous adaptation process is a process that can tolerate such delays according to the definition in our taxonomy and so this is not an issue.

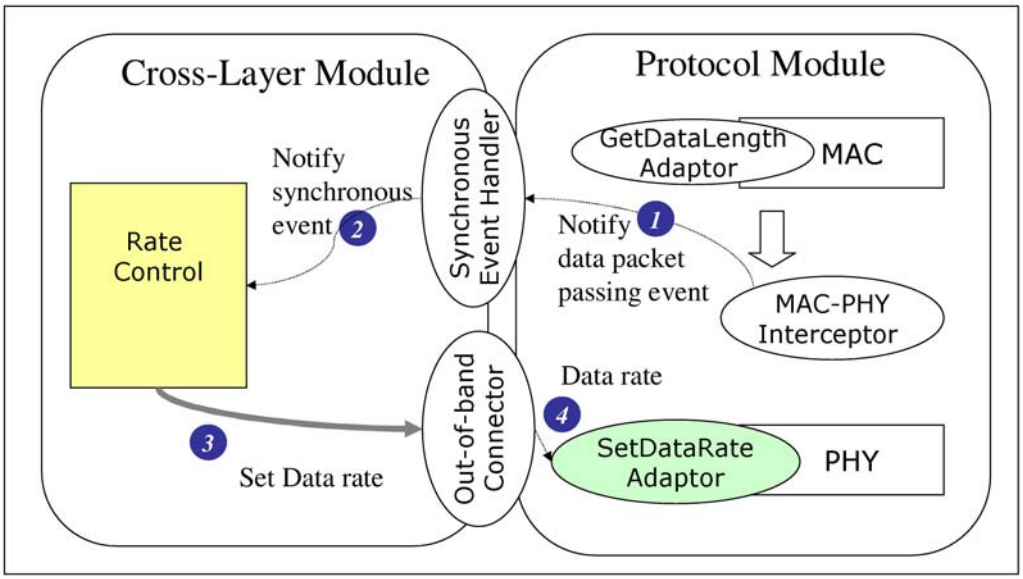

Figure 13: Reusing the components for the rest of process

3.6 COMPLETING THE ARCHITECTURE

Figure 14 shows all the details of our architecture. Most aspects of this diagram have already been presented, the main refinement is in the connectors presented and their relation to whether the cross-layer processor is synchronous or asynchronous. These all are typical software connectors. Disregarding the event handler aspect for now, we see four kinds of connectors, corresponding to the four delivery mechanisms in the taxonomy. The Intranode connectors are used inside a single node to integrate existing protocol modules with our architecture. The In-band connector accesses the data stored in a packet's internal structure when the packet passes though an interceptor, while the Out-

of-band connector uses adaptors to access data in the protocol modules themselves. The Internode connectors require that any information must be placed in a packet and sent from one node to another. In the In-band case the information can be piggybacked on the protocol packet. Thus this case is shown intercepting the data in the packet delivery path. In the Out-of-band case, the information must be formatted into its own packet and sent independently. Thus this is shown as a separate communication path. Returning to the event handlers, we see that the asynchronous versus synchronous nature of the processes is fundamentally captured by the type of the event handler. Synchronous event handlers are driven by the passage of packets through the interceptors. However, Asynchronous events (and thus processes) are triggered when the event handler inside Intranode version of Out-of-band connector detects a change of information within a protocol processor or when the Internode version of event handler receives a new information from counterpart in another node.

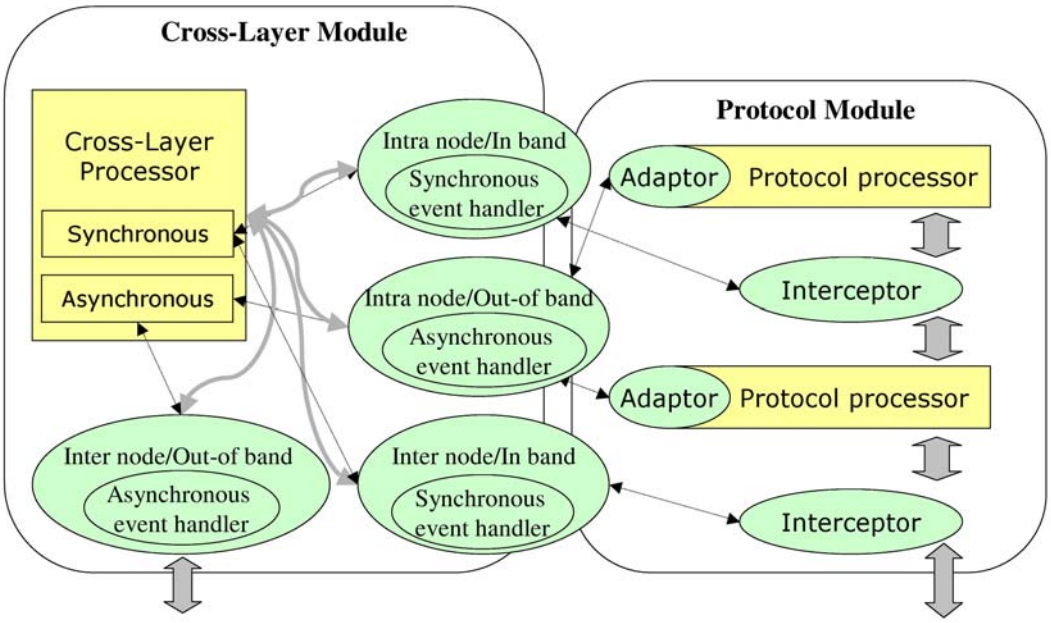

Figure 14: A conceptual architecture cross-layer wireless network adaptations

3.7 Validation

Initial validation of our taxonomy and architecture was done by careful consideration and paper design of the examples found earlier, as well as others. A more substantial validation is ongoing using the basic strategy of implementing our framework and a number of our examples in a realistic wireless network testbed. We expect this

experience to allow us to refine our approach, in particular with respect to what concrete architectures are desirable.

3.8 HYDRA

Our implementation has been done in the context of our Hydra testbed. Hydra is a prototype multihop wireless network, which is designed to allow experimentation with implementations of PHYs, MACs, and cross-layer adaptations, using functional hardware and software, rather than simulation. Hydra uses an RF frontend, the universal software radio peripheral (USRP) from Ettus Research, which allows experimentation with various RF frequency bands. The USRP connects to the Hydra PHY over USB 2.0. The PHY is implemented using the GNU Radio framework and all signal processing is done using the general-purpose processor. Hydra's MAC interfaces to the PHY using interprocess communication and is implemented using the Click modular router. Click also provides network support and interfaces to the Linux TCP/IP stack allowing full end-to-end application-to-application experiments. Both GNU Radio and Click allow us to implement flexible network protocols. Using the GNU Radio framework, we create a set of ``signal processing blocks'', each of which implements a signal-processing algorithm. Then we can compose a PHY protocol by connecting these small blocks into a signal processing flow graph. Similarly, Click allows us to create a set of ``packet processing elements'', each of which implements some task required for packet processing and to compose a new protocol by connecting the elements.

Hydra is currently operational. The current Hydra implementation is similar to 802.11 PHY. It supports orthogonal frequency division multiplexing (OFDM) and multiple input multiple output (MIMO) at the physical layer, with support for multiple transmission rates. The MAC is essentially the 802.11 DCF MAC briefly discussed earlier.

3.9 A CONCRETE ARCHITECTURE FOR HYDRA

We refined our conceptual architecture to implement it in Hydra. The key challenge was that protocols in Hydra are implemented using three different protocol modules. The MAC and Network layers are implemented using Click while the PHY uses GNU Radio. Further the TCP/IP protocol stack implements the Transport layer. This meant that the Interceptors and Adaptors needed to be implemented differently to conform to each implementation environment. Such a protocol configuration that is composed of multiple protocol modules is generally true for most wireless system. For example, the wireless MAC protocol is implemented on a network interface card while routing protocols are implemented using the TCP/IP protocol stack. To address this problem, we divided all of the connectors into two levels, except the Internode version of the Out-of-band connector that is not connected to the protocol modules. Each local connector is implemented conforming to implementation environment provided by each protocol module and thus easily manages the Interceptors and Adaptors that are implemented using the same environment. Then the local connectors communicate with global connectors that provide cross-layer processors with event notification and data delivery. Thus this concrete

architecture still allows the cross-layer adaptations to be independent of the existing protocol implementations.

3.10 RATE ADAPTATION

We have implemented both Intranode and Internode rate control using our fully decoupled architecture. We have also implemented both versions in the ad-hoc manner that might be considered ``conventional" to compare both the implementation techniques.

The conventional implementation required a set of changes to the existing MAC and PHY implementations. To implement both the Intranode and Internode version of rate control processes:

1. A set of Click elements were created and modified: - to allow the MAC to obtain the channel status information and to execute rate adaptation - to use the new CTS packet format that delivers the channel state from the receiver to the transmitter.
2. A GNU Radio block was changed: - to allow the MAC to access the channel status.
3. The interfaces between the MAC and PHY were changed: - to allow cross-layer information piggybacked on packets to be marshalled and unmarshalled when they move between the MAC and PHY.

The key problem was that such changes cause individual protocol processors to become dependent on others. At one point we changed the rate control in the MAC to use a different type of channel information, from an integer valued received signal strength indication (RSSI) to a floating point valued SNR. This required that the interfaces between the MAC and PHY and the signal processing block in the PHY all needed to change to deal with the new type of channel information.

We then implemented the rate control processes based on our loosely coupled architecture. These implementations were encouraging in that they did not introduce any significant changes into the existing protocol processors. After implementation of the global and local connectors, to implement both the Intranode and Internode version of rate control processes:

1. A set of Click elements were created: - to add the Interceptors into the MAC - to attach an Adaptor to access data length
2. A GNU Radio block was created: - to attach an Adaptor that accesses channel status.
3. Rate control processor was created: - to execute rate control.

Although a set of protocol processors were created and inserted into Click and GNU Radio to implement our architectural components, these components did not introduce any significant changes in the existing protocol implementations. An Interceptor notifies the Synchronous event handler of the passage of a CTS packet and transparently changes the format of the CTS packet. Further the Adaptors augmented the interfaces of the MAC and the PHY without affecting core functionality of the existing protocol processors. The only change caused by the Adaptor was to expose some variables inside the protocol processors to allow the Adaptor to access the information.

We implemented the Internode version of rate control by extending the Intranode version. To extend the adaptation process in a conventional implementation, we needed to modify a few more packet processing elements in Click. The problem was that the dependency between the Click elements changed. However, the implementation based on our architecture only required extension of the existing rate control processor after inserting a few more Interceptors into Click. This shows that our architecture allows rate control to be independent of the infrastructure and to freely change its operation without significant impact on existing protocol implementations.

4.0 Area 3: Abstractions that allow developers to express complex communication requirements in dynamic wireless environments

4.1 The Scene Abstraction

The final thrust of this project investigated software abstractions that simplify the development of applications in dynamic wireless environments. We have developed the *scene* abstraction [6, 7], which enables an application developer to declaratively define a region of a wireless network with which to interact. To create a scene tailored to his application, a developer writes a declarative specification encompassing metrics over environmental, network, and device characteristics. Given these metrics and a corresponding threshold, we can dynamically construct a distributed data structure that represents a particular application's scene. As a device moves and its operating environment changes, its scene automatically updates to reflect these changing conditions. Once a scene is defined, an application can subsequently use the dynamically changing data structure to interact with exactly the portion of the operating environment that impacts its behavior. Through simulation, we have shown that a naïve protocol that performs a distributed computation can build and maintain scenes with relatively low overhead [6]. The same measurements have also demonstrated that our approach scales well for increasing scene size and client mobility. We have also designed a middleware architecture around the scene approach [8]. The DAIS (Declarative Applications in Immersive Sensor Networks) middleware defines a hierarchical architecture in which more powerful devices (i.e., client devices) execute significant middleware chunks that

interact with slimmed-down middleware components that execute on significantly resource-constrained embedded sensors. The scene abstraction provides a fundamental interactive capability for the DAIS middleware that revolutionizes the manner in which networked devices can interact to support applications' requirements. In this report, we summarize the key aspects of the scene abstraction that have appeared in [6, 7]

4.2 Motivating a New Network Abstraction

In immersive networks, applications commonly need to interact directly with devices embedded in the environment. This allows applications to operate over information collected directly from the local area (as shown in Figure 15). This is in contrast to existing deployments in which the networks are commonly accessed through a single collection point. While existing approaches that sense, aggregate, and stream information to a central collection point may also be useful to applications, we focus on enabling applications' direct, on-demand, mobile interactions. This new style of interaction is a direct motivation for a reexploration of protocol and coordination issues in immersive networks and embodies several unique challenges:

- *Locality of interactions*: An application interacts directly with local devices, which can minimize communication overhead and latency. However, such a direct interaction approach can also be cumbersome with respect to enabling the application to precisely specify the area from which it collects information.

- *Mobility-induced dynamics*: The application interacting with the network runs on a device carried by a mobile user, and the devices in the network may also be mobile. Devices' interconnections and the area from which an application draws information are subject to constant change.

- *Unpredictability of coordination*: We focus on using general-purpose networks for dynamic mobile computing applications. Therefore, few *a priori* assumptions can be made about the applications' needs or intentions, requiring networks to adapt to unexpected and changing situations.

The confluence of these challenges necessitates the development of a new paradigm of communication for applications in dynamic wireless environments.

4.3 Scenes: Declarative Local Interactions

The set of data sources near a user changes based on the user's mobility. If the network is well connected, the user will be able to reach vast amounts of raw information. The application must limit the scope of its interactions to only the data that matches its needs. In our model, an application's operating environment is encapsulated in a *scene* that constrains which networked devices influence the application. This abstraction defines local, multihop neighborhoods surrounding a particular application, supports mobility by dynamically updating the *scene*'s participants, and minimizes how much a developer

must know about underlying implementations. The constraints that define a *scene* may be on properties of hosts (e.g., battery life), network links (e.g., bandwidth), and data (e.g., type).

Defining Scenes

The declarative specification defining a *scene* allows an application programmer to flexibly describe the type of *scene* he wants to create. Multiple constraints can be used to define a single *scene*. The programmer needs only to specify three parameters to define a constraint:

- *Metric*: A property of the network or environment that defines the cost of a connection (i.e., a property of hosts, links, or data).
- *Path cost function*: A function (such as sum, average, minimum, or maximum) that operates on a network path to calculate the cost of the path.
- *Threshold*: The value a path's cost must satisfy for that sensor to be a member of the *scene*.

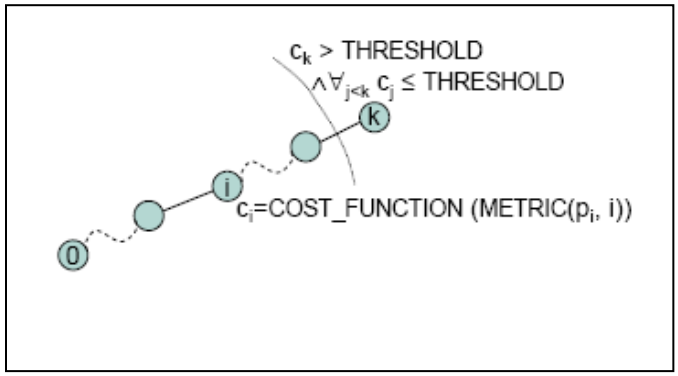

Figure 15: Distributed *scene* computation

Figure 15 demonstrates the relationships between these components. This figure shows only a one-constraint *scene* and a single network path. The cost to a node in the path (e.g., node *i*) is calculated by applying the path cost function to the metric using information about the path so far (p_i) and information about this node. Nodes along a path continue to be included in the *scene* until the path reaches a node whose costs (e.g., c_k) is greater than the *scene*'s threshold. This functionality is implemented in a dynamic distributed algorithm that can calculate (and dynamically recalculate) *scene* membership.

The selected network paths correspond to branches of a routing tree created as part of *scene* construction. When a node needs to relay a reply back to the user, the reverse of the path on the routing tree can be used. If a node receives a *scene* message that it has already processed, and the new metric value is shorter, this path is chosen, and the node forwards the information again because it may enable new nodes to be included in the *scene*.

The *scene* definition can be formalized in the following way:

> Given a client α, a metric M, and a positive threshold, T, find the set of hosts S_α such that all hosts in S_α are reachable from α and, for all hosts β in S_α, the cost of applying M on some path from α to β is less than T:
>
> $$S_\alpha = \langle set\ \beta : M(\alpha,\beta) < T :: \beta \rangle$$ [1]

The above formalization is a simplification of the *scene* formalization; it uses only one metric. If the *scene* is specified by multiple metrics, the center expression must be true for all metric/threshold pairs.

Maintaining Scenes

While a *scene* provides the appearance of a dynamic data structure, the implementation behaves on demand; no proactive behavior occurs. Only when the application uses a *scene* does the protocol communicate with other local devices, reducing the overall communication overhead. At first glance, this approach may appear to incur an unpredictable latency for the first query posed to a *scene*. However, queries traverse the same path as the *scene* construction messages, and the queries themselves carry the *scene* construction information. Therefore, the on-demand construction incurs no additional latency.

For one-time queries, a *scene* is created, and the scene information is not stored or updated in any way. However, if the *scene* is to be used for a persistent query, it needs to be maintained. To maintain the *scene* for such continuous queries, each member sends periodic beacons advertising its current value for the metric. Each node also monitors beacons from its parent in the routing tree, whose identity is provided as previous hop

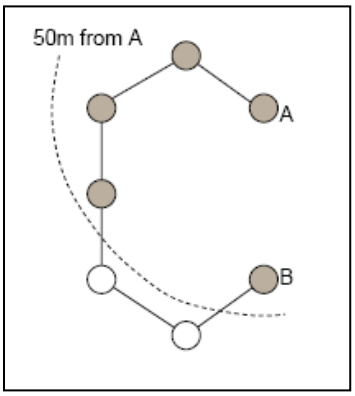

Figure 16: A C-network

[1] In the three-part notation: ⟨*op quantified_variables* : *range* :: *expression*⟩, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a mulitset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, then the value of the three-part expression is the identity element for *op* (e.g., ∅ if *op* is *set*).

information in the original *scene* message. If a node has not heard from its parent for three consecutive beacon intervals, it disqualifies itself from the *scene*. This corresponds to the node falling outside of the span of the *scene* due to client mobility or other dynamics. In addition, if the client's motion necessitates a new node to suddenly become a member of the *scene*, this new node becomes aware of this condition through the beacon it receives from a current *scene* member.

Defining Scenes Based on Physical Characteristics

The metrics used to specify *scenes* can be divided into two categories: those that define *scenes* based on properties of network paths or the devices on the network paths (e.g., latency or battery power) and those that define *scenes* based on physical characteristics of the environment (e.g., location or temperature).

Using a physical characteristic to calculate network paths is plagued by the C-shaped network problem. Consider the network shown in Figure 16. Nodes A and B are within 50m of each other, yet a discovery from A to B must leave the region of radius 50m surrounding A to find B. The only way to guarantee that every device within 50m is discovered is to flood the entire network.

In the types of applications we address, an application may not be in direct communication with the devices with which it needs to interact. For this reason, we focus on building the best multi-hop neighborhoods possible. For metrics that measure physical characteristics, the question remains as to how to handle the ambiguity separating the natural specification (e.g., "all devices within 50m") and the ability of a protocol to efficiently satisfy that specification (i.e., without flooding the network). Given our experience with the complexity involved in creating region specifications, we favor an approach that does not require strictly increasing metrics. This makes the programming interface simpler, but in the presence of configurations like that shown in Figure 16, our approach may not find some members of the specified *scene* even though they are transitively connected.

The results demonstrate that our approach tends to find the vast majority of the *scene* members under reasonable conditions. Therefore, we favor natural *scene* specifications over complete accuracy of scene membership.

4.4 Realizing Scenes on Resource-Constrained Devices

The model for both construction and maintenance of *scenes* described in the previous section is tailored to the requirements of resource-constrained environments. In creating applications for client devices, developers can leverage a Java interface, which allows them to use the *scene* communication abstraction to interface with other devices embedded in the environment. Software on these devices must also support the *scene* abstraction. In this section, we describe this implementation, showing how the code is structured to support dynamic, opportunistic communication.

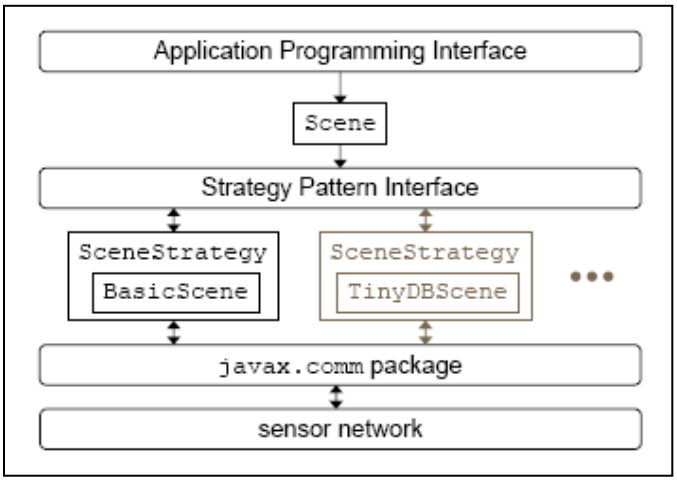

Figure 17: Simplified software architecture

A Structured Implementation Strategy

Our implementation uses the strategy pattern, a software design pattern in which algorithms (such as strategies for *scene* construction and maintenance) can be chosen at runtime depending on system conditions. The strategy pattern provides a means to define a family of algorithms, encapsulate each one, and make them interchangeable. Such an approach allows the algorithms to vary independently from the clients that use them. In the *scene*, the clients that employ the strategies are the queries, and the different strategies are `SceneStrategy` algorithms. Figure 17 shows the resulting architecture. We decouple the *scene* construction from the code that implements it so that we can vary message dissemination without modifying application-level query processing (and vice versa).

The remainder of this section describes one implementation of the `SceneStrategy`, the `BasicScene`, which provides a prototype of the protocol's functionality. Other communication styles can be swapped in for the `BasicScene`. By defining the `SceneStrategy` interface, we enable developers who are experts in existing communication approaches to create simple plug-ins that use different query communication protocols and yet still take advantage of the *scene* abstraction and its simplified programming interface.

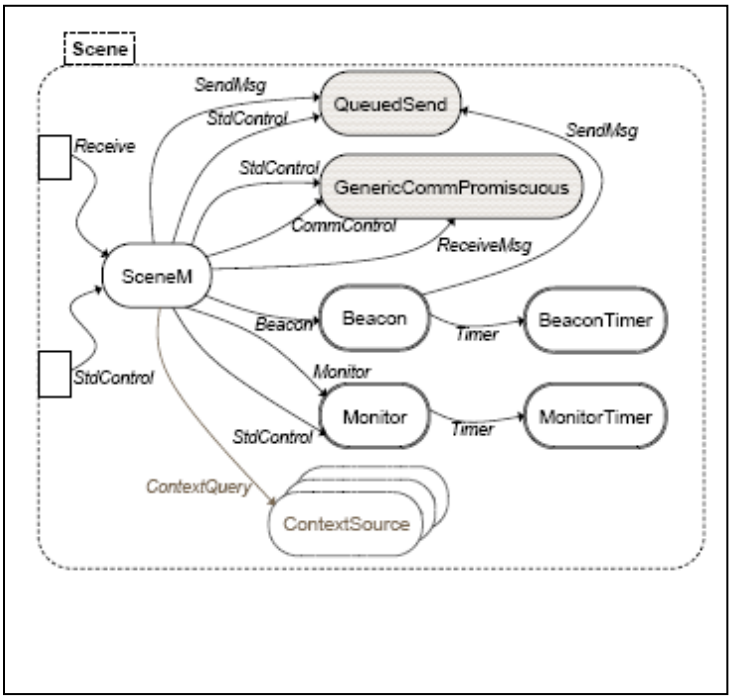

Figure 18: **Implementation of the scene in nesC**

A Basic Instantiation

While the *scene* abstraction is independent of the particular hardware used to support it, in our initial implementation, these software components have been developed for Crossbow Mica2 motes and are written for TinyOS in the nesC language. Our nesC implementation of the *scene* abstraction (along with other project information) is available at http://mpc.ece.utexas.edu/scenes/index.html. In nesC, an application consists of modules wired together via shared interfaces to form configurations. Figure 18 depicts the components of the scene configuration and the interfaces they share.

This implementation functions as a routing component on each node, receiving each incoming message and processing it as our protocol dictates. In this picture, we show components as rounded rectangles and interfaces as arrows connecting components. A component provides an interface if the corresponding arrow points toward it and uses an interface if the corresponding arrow points away it. If a component provides an interface, it must implement all of the commands specified by the interface, and if a component uses an interface, it can call any commands declared in the interface and must handle all events generated by the interface.

The `Scene` configuration uses the `ReceiveMsg` interface (provided in TinyOS), which allows the component to receive incoming messages from the radio (by handling the receive event). Specifically, within the Scene configuration, the `SceneM` component

handles this event. SceneM implements most of the logic of the scene implementation on the sensor.

While the model allows a *scene* to be defined by multiple constraints, a single SceneMsg contains only one constraint. This is a limitation of our proof-of concept implementation that will be removed in a more mature implementation. The SceneMsg contains a sequence number that uniquely identifies the message. The sequence number is a combination of the unique client device id and the device's sequence number. This allows a receiving node to differentiate between scenes for different client applications. A message contains two constants that instruct the SceneM component in processing the message: the metric and the path cost function. The use of constants to specify the metric and cost function makes the implementation a little inflexible because the set of metrics must be known *a priori*, but the approach prevents messages from having to carry code. Future work will enable this automatic code deployment. The metricValue in the SceneMsg carries the previous node's calculated value for the specified metric and is updated at the receiving node. In the case of a *scene* based on location, the metricValue may be the location of the source node, while in the case of a metric based on end-to-end latency, the metricValue may be the aggregate total latency on the path the message has traveled. The previousHop in the SceneMsg allows this node to know its parent in the routing tree and enables *scene* maintenance. The SceneMsg's maintain flag indicates if the query is long-lived (and therefore whether or not the *scene* should be maintained). Finally, each SceneMsg also carries the query to deliver to the application.

Figure 21 shows how a *scene* message is processed at a receiving node. When SceneM receives a message it has not received before (based on the message's unique sequence number), it determines whether the node should be a member of the *scene* by calculating the node's metric value based on the metric and path cost function. Because these fields are constants, SceneM can look up their meanings in a table and determine how to calculate the new metric value. Depending on the metric, this may require ContextSources, which provide relevant data for calculating a node's value. If necessary, context values are retrieved from the designated ContextSource through the query command in the ContextQuery interface. If the metric demands a context source that the node does not provide, the device is not considered a part of the *scene*. When the necessary context values have been retrieved, the costFunction from the message is invoked. The newly calculated value for the metric is compared against the value of the threshold in the SceneMsg. If the new value does not satisfy the threshold, then this node is not within the *scene* and the message is ignored.

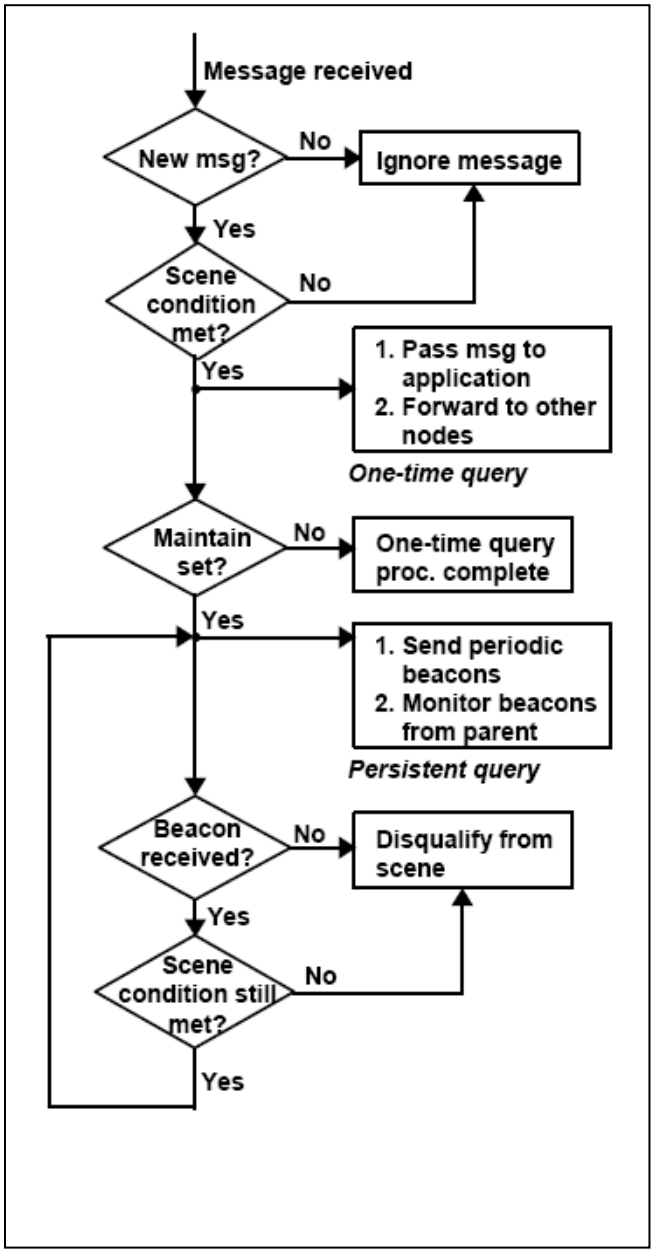

Figure 19: Scene construction flow chart

If this node is within the *scene*, the message is forwarded to allow inclusion of additional nodes. The node replaces the `previousHop` field with its node id. The `metricValue` field is populated according to the type of the metric. This new message is broadcast to all neighbors (using TOS_BROADCAST_ADDR as the destination). The node also passes data to the application (through the `Receive` interface).

The *scene* also needs to be maintained in the case of a persistent query. If the maintain flag is set, then `SceneM` must monitor changes that may impact the node's membership. For example, if the *scene* is defined by relative location and the user is walking through the network, as he moves away from a sensor, the sensor will need to be removed from the *scene*. The *scene* implementation on the sensors uses a `Beacon` module to transmit periodically to other nodes. As the `Monitor` component (described next) detects changes in the metric value, the value is updated (through `SceneM`) and reflected in the beacons sent to neighbors. In addition, `SceneM` must monitor incoming beacon messages from the parent. Such messages are received in `SceneM` and passed to the `Monitor`. The `Monitor` uses beacons from the parent, information about the *scene* (from the initial message), and information from the context sources to monitor whether the node remains in the scene. In addition, the `MonitorTimer` requires that the node has heard a beacon from the parent at least once in the last three beacon intervals. If either the parent has not been heard from or the received beacon pushes the node out of the scene, the `Monitor` generates an event for `SceneM` that ultimately ceases the node's participation in the *scene*, including signaling the application to cancel its interactions with the client device.

4.5 Evaluation and Analysis

In this section, we provide an evaluation of our implementation. Our protocol's intent and behavior differ significantly from other approaches, so direct comparison to existing protocols is not very meaningful. Instead, we measured our protocol's overhead in varying scenarios, by employing TOSSIM, a simulator that allows direct simulation of code written for TinyOS. TOSSIM therefore allows us to perform large-scale simulations (in this case, of 100 nodes); these simulations are of a scale that is unmanageable in real sensor networks. However, the code executed in TOSSIM is the same code running on the sensors, and small-scale runs in the sensors themselves corroborate the results reported here.

Simulation Settings

In generating the following results, we used networks of 100 nodes, distributed in a 200 x 200 foot area, with a single client device moving among them. We used two types of topologies: 1) a regular grid pattern with 20 foot internode spacing and 2) a uniform random placement. While the sensor nodes remained stationary, the client moved among them according to the random waypoint mobility model with a fixed pause time of 0. To model radio connectivity of the nodes, we used TOSSIM's empirical radio model, a probabilistic model based on measurements taken from real Mica motes. In all cases, as the client moves, the *scene* it defines updates accordingly. In the different simulations, the client either remains stationary or moves at 2mph, 4mph, or 8mph. While these speeds appear to be on the slow side, they are reasonable for the pervasive computing scenarios we consider (e.g., 4mph is a very brisk walk; 8mph is an expected speed of vehicles on construction sites, etc.). In these examples, *scenes* are defined based on the number of hops relative to the client device, ranging from one to three hops. Other

metrics can be easily exchanged for hop count; we selected it as an initial test due to its simplicity.

A final important parameter in these measurements is the beacon interval. Recall that the beacon interval is the specified amount of time over which each node monitors beacons from its parent in the routing tree to maintain its own membership in the scene for continuous queries. If the node does not hear from its parent during that beacon interval, it disqualifies itself from the scene. We have currently set the beacon interval to be inversely proportional to client speed. Because this approach relies on shared global knowledge, this is not how beacon intervals will actually be assigned, and future work will investigate better ways of assigning this value. For example, in the future, clients can monitor their own speeds and embed this beacon interval in *scene* building packets.

Alternatively, individual sensors could monitor the change in client connections over time (which can be correlated with mobility) and use this information to locally adapt the beacon interval. Both options allow nodes to adapt the beacon interval depending on the particular situation; this adaptation is critical to context.

Performance Metrics

We have chosen three performance metrics to evaluate our implementation: (i) the average number of *scene* members, (ii) the number of messages sent per *scene* member, and (iii) the number of messages sent per unit time. We evaluate these metrics for both grid and random topologies.

The first metric measures how well our selected beacon intervals perform. The latter two metrics measure the scalability of the *scene* abstraction, i.e., how the protocol will function in *scenes* of increasing sizes and client mobility. The number of messages sent per *scene* member measures a sensor node's cost of participation, which also estimates the potential battery dissipation for the sensors that participate in the *scene* (since energy expended is proportional to radio activity). The number of messages sent per unit time is a measure of the network's average activity. Since the *scene* protocol operates on demand, activity takes place only within the *scene*.

Simulation Results

The number of *scene* members is almost independent of the client node's speed. This means that the device is able to accurately reach the nodes that need to be members of its *scene* and shows that setting the beacon frequency to be proportional to the client node's speed accurately keeps track of the moving client.

Because we have set the beacon frequency to be directly proportional to the speed of the client node (e.g., if the client speed is 4 mph, beacons are sent every 0.5s, if the client speed is 8 mph, beacons are sent every 0.25s), beacons are sent more frequently as speed

increases, yielding the linear relationship. This shows how the battery dissipation for each sensor that participates in the *scene* would scale with increasing client mobility.

Beacons are sent more frequently as the client node speed increases, causing more messages to be packed into a given time interval. In addition, as the *scene* size increases, more nodes become *scene* members, increasing the number of nodes that subsequently send beacon messages per unit time.

These results demonstrate that even as the scene size increases, the overhead of creating a local communication neighborhood is manageable and localized to a particular region of interest. Since the *scene* protocol is an on-demand communication protocol, the activity in the network takes place only within the *scene*. The nodes that do not satisfy the *scene* constraints are inactive. The average number of *scene* members stays constant over changing client mobility for a specified *scene* size.

4.6 Discussion and Future Work

The evaluation reported in the previous section demonstrates the feasibility of placing a data communication abstraction for pervasive computing on highly resource-constrained devices. This is an important step as such sensing devices are an essential part of any immersive sensor network that supports such applications. The quantitative evaluation in the previous section, however, does not address the relationship of the *scene* abstraction to other neighborhood protocols. This is largely due to the fact that the client-centered, device-agnostic form of the *scene* abstraction provides a completely novel perspective on interactions in sensor networks. More qualitative comparative evaluations are planned as future work.

Another important point of evaluation not addressed in this paper is that of the expressiveness of the *scene* abstraction. Specifically, future work will investigate the question of how rich and usable the abstraction is with respect to the requirements of a variety of applications. Our own work with intelligent construction sites [9] has demonstrated applicability to a second domain (other than the first responder domain used in this paper), but we plan to perform additional user and application studies to further validate our expressiveness claims. In addition, we have made our implementation available at http://mpc.ece.utexas.edu/scenes/index.html for others to download and try for their applications.

A final interesting point of discussion is that of the degree of adaptivity the *scene* abstraction provides. We have motivated throughout that awareness of and adaptation to the surrounding environment are crucial to enabling pervasive computing applications. The *scene* abstraction as described already incorporates several points of adaptation, most specifically allowing the participants in a *scene* to change over time in response to client mobility or to changes in the network or physical environment. We have already discussed such adaptation with respect to setting the *scene*'s beacon frequency to be sensitive to the client mobility or a sensor node's local perception of mobility. Future work will explore additional adaptation points that could make the abstraction even more

responsive to pervasive computing applications. For instance, one could imagine a *scene*'s threshold expanding or contracting based on the environmental values sensed or the density of available readings.

4.7 References

[1] K. Mandke, S. Choi, G. Kim, R. Grant, R. C. Daniels, W. Kim, R. W. Heath, Jr., and S. Nettles, ``Early Results on Hydra: A Flexible MAC/PHY Multihop Testbed," *in the Proc. of IEEE Vehicular Tech. Conf.*, Dublin, Ireland, April 23 - 25, 2007

[2] K. Mandke, R. C. Daniels, S. Choi, R. W. Heath, Jr., and S. Nettles, ``Physical Concerns for Cross-Layer Prototyping and Wireless Network Experimentation," *In the Proc. of the Second ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization (WiNTECH)*, Montreal, Canada, September 10, 2007

[3] Soon-Hyeok Choi, Dewayne E. Perry and Scott M. Nettles, "A Software Architecture for Cross-Layer Wireless Network Adaptations", *to appear in the Working IEEE/IFIP Conference on Software Architecture (WICSA) 2008*, Vancouver, BC, Canada, February 2008.

[4] http://users.ece.utexas.edu/~perry/work/papers/060908-SC-adaptations.pdf

[5] S. Choi, R. Grant, W. Kim, H. Wright, R. W. Heath, Jr., and S. Nettles, ``An Experimental Evaluation of Several Rate Adaptation Protocols," In preparation.

[6] S. Kabadayı and C. Julien. A Local Data Abstraction and Communication Paradigm for Pervasive Computing, in *Proceedings of the 5th Annual IEEE Conference on Pervasive Computing and Communications (PerCom)*, March 2007, pp. 57—66.

[7] S. Kabadayı, and C. Julien, "Scenes: Abstracting Interaction in Immersive Sensor Networks," *Pervasive and Mobile Computing*, 3(6):607—738, 2007.

[8] C. Julien, and S. Kabadayı, "Enabling Programmable Ubiquitous Computing Environments: A Middleware Perspective," *Advances in Ubiquitous Computing: Future Paradigms and Directions*, 2007 (to appear).

[9] J. Hammer, I. Hassan, C. Julien, S Kabadayı, W. O'Brien, and J. Trujillo. "Dynamic Decision Support in Direct-Access Sensor Networks: A Demonstration," in *Proceedings of the 3rd International Conference on Mobile Ad-hoc and Sensor Systems (MASS)*, 2006, pp. 578—581.